information SERVICES

# Unix 3
# Shell Programming

Workbook

December 2016
Document Reference: 2630-2016

# Contents



If you require this document in an alternative format, such as large print, please email is.skills@ed.ac.uk.

# 1. Shell Programming

## Section 1

# The Philosophy

- **Unix is a rich programming environment**
- **small tools for specific tasks**
- **shell is glue!**

### Unix history

Unix was originally developed in AT&T Bell Labs in the late 1960s. It was written at that time as a development environment for programmers within AT& T . As such, a programmable command language which interfaces to the many system tools was at the heart of Unix's design.

### Small + shell = big

The philosophy for program development under Unix at that time was small tools for specific tasks. These small tools could be harnessed using the shell's input/output redirection mechanisms ( eg pipes ) to form larger tools. For example, by providing a "sort" command the output of any other command can be ordered as required by piping its input into "sort". So the task of sorting a command's results is done by a tool whose only task is to sort. The original command does not need to know how to sort. As Unix has developed over the years many tools have appeared which rebel against this philosophy; no one would call GNU emacs a small tool! This philosophy is however still very important.

### Sticky Shell

The shell provides the glue to construct larger commands from more specific tools. The same shell which is used as an interactive command

interpreter is the shell used as a programming language. Not only does the shell provide the input/output redirection which allows these tools to be linked together, it also has a rich programming syntax. The shell provides variables, conditional expressions ( ie if statements ) loops and subroutines,

## Task 1.1 Find commands that are scripts

A number of commands available in Unix are shell scripts themselves. Use the "file" command on a system bin directory to determine a file's contents and pipe its result into grep to look for the string "commands text", eg

```
bash$ file /bin/* | grep script
```

All these commands are shell scripts.

# The Bourne Shell

- **Why the Bourne shell?**
- **Bourne shell syntax works for bash and ksh**
- **Bash (Bourne again shell) available as standard on modern Linux systems**
- **Other shells - csh tcsh ...**

---

### The Bourne Shell

The shell covered in this course is the Bourne shell. The Bourne shell is the original Unix shell developed by Stephen Bourne. The other popular Unix shell was the C shell; a shell which tries to mimic the C programming language syntax. The Bourne shell is available under all Unix systems and is generally considered to have a much clearer syntax.

### Other shells

All Unix shells roughly fall into two camps; those derived from the Bourne shell and those derived from the C shell. The programming syntax described in this course will work with those shells in the Bourne shell camp. The most common shells in this camp are the Korn shell (ksh) and the Bourne again shell (bash) which are the most widely used Unix shells. The csh camp contains the "tenex "style shells. The Bourne shell programming syntax will not work with the C shell or those derived from it, though the concepts described in this course are still relevant.

Any references to "the shell" in this document are thus references to the Bourne shell.

## Task 1.2 Compare shell and nano sizes

The Bourne shell is stored in the file /bin/sh. Do an ls -l of this file and look at its size.

A version of the nano editor is stored in /usr//bin/nano. Again, do an ls -l and look at its size.

Though relatively small the Bourne shell is very powerful.

## Revision – Shell metacharacters

- **Asterisk** ***** **any sequence of characters**
- **Square brackets** **[abc]** **the specified range**
- **Tilde (bash only)** **~** **home directory**
- **Dollar symbol** **$** **dereference variable**

### Shell metacharacters

Shell metacharacters are special characters that provide a short-hand for specifying filenames and, as described later, matching strings.

### Asterisk

The asterisk matches any string of characters including the empty string.

For example -

a* will match any string starting with an "a" including the string "a" itself.

### Square brackets

Square brackets indicate sets of characters and will match a single character which is in the set. Two characters separated by a dash match any character lexigraphically between the pair. More than one range can be used within the square brackets.

For example:

[abc] will match any of a b c

[a-c] will match any of a b c

[a-cA-c] will match any of a b c A B C

Square brackets are most commonly used with asterisks to match strings starting with a specific range.

For example: [a-cA-C]* will match any string starting with a b c A B C

### Tilde

The tilde character will match the specified user's home directory or if no user is specified, the home directory of the user executing the command. Note that this facility is only present when using the bash shell and not when using the standard Bourne shell.

For example: ls ~barney will attempt to list the contents of the "barney" user's home directory.

### Dollar symbol

The dollar symbol will expand (dereference) the value of a variable.

For example: echo $PATH will output the contents of the PATH variable to the screen.

# Revision - Shell quoting

- **Quoting - protect characters with special meaning to shell**
- **Backslash \ shields (escapes) one character**
- **Single quotes '...' shields all enclosed characters ( except ' )**
- **Double quotes "..." shields enclosed characters except for $ ` " **

---

### Quoting

Quoting is used to shield the shell's metacharacters from interpretation.

### Backslash

The backslash will shield any single character it precedes.

### Single forward quote

All characters enclosed between a pair of single forward quotes are shielded - apart from the ' character itself!

### Double quotes

All the characters enclosed between a pair of double quotes are shielded except for $ ` \ and "

# Revision - redirection

- **Pipes |**
- **Output redirection**
  - **>**
  - **>>**
- **Input redirection**
  - **<**
- **Here Documents**
  **<< EOF**
  **text**
  **EOF**

---

### Pipes

Pipes join the output of the last command to the input of the next, eg

```
bash$ who | wc -l
```

would pipe the output of the "who" command into the wc command, and so counts how many users are logged in.

### Output redirection

The > character sends the output of the command into a named file.

For example -

```
bash$ who > wholist
```

will create or overwrite a file which contains a list of who is on the system.

The >> character appends the output of the command into the named file.

For example -

```
bash$ who >> wholist
```

will append a list of who is on the system to the end of the file wholist.

### Input redirection

The < character takes its input from the named file.

For example-

```
mail tony < /etc/motd
```

will send the message of the day ( /etc/motd ) by email to user tony.

### Here Documents

The text between the delimiters is used as the standard input of the command. Between the delimiters shell variable expansion and command substitution are performed.

For example,

```
cat << ENDSTRING
You are about to enter an interactive session which will modify the
behaviour of this system to suit your use.
ENDSTRING
```

will display the text between the delimiters to the screen.

Here documents are useful for displaying known multi-line text to the screen without having to create a separate file to contain the text.

# Revision - command substitution

- **Command substitution**
  - **`command`**

```
echo “Today’s date is `date`”
Today’s date is Thu 7 Jul 2016 16:53:20 BST
```

- **$( …. ) in bash only**

```
echo “Today’s date is $(date)”
Today’s date is Thu 7 Jul 2016 16:53:31 BST
```

### Command substitution

Commands enclosed by backquotes are run by the shell and the text enclosed by and including the backquotes are replaced by the result of the command. Newlines in the output from the command are replaced by spaces.

The bash shell has an alternative syntax of $(command). This can make nesting multiple layers of command substitution easier and more legible.

For example-

```
bash$ echo "Next year will be $(expr $(date +%Y) + 1)."
Next year will be 2017
```

In this example we’re using the date +Y command to get the current year then using the expr command to add 1 to the result.

### The shell

Command substitution is good for presenting arguments to commands eg

```
bash$ ls -l `which ls` -r-xr-xr-x 1 root bin 100440 Feb 17 1994 /bin/ls
-rwxr-x--x 1 root root 144881 Jan 13 1992 /usr/local/gnu/bin/ls
```

or for assigning shell variables, eg

```
bash$ Date=`date` bash$ echo $Date Thu Jun 8 12:00:28 BST 1995
```

The shell actually evaluates the commands between the backquotes by executing a new shell and so any shell constructs can be used within the backquotes.

For example -

```
bash$ Numusers=`who | wc -l` bash$ echo $Numusers 250
```

# Revision- Regexps and Filters

- **Regular expressions identify text patterns in files**
- **Filters operate on a stream of text**
- **grep,awk,sed,sort are examples of filters**
- **Filters can be combined using pipes “|”**

---

### Filters

Filters accept text from standard input and output it via standard output which means they can be combined together using redirection operators. This is a good example of the Unix philosophy of having small tools that can be joined together to perform a more complex task. The filter commands are discussed in greater detail in the “Power UNIX” section of the UNIX 2 course booklet.

### grep

Grep is the General Regular Expression Parser and it will print any line in a file which matches a given regular expression e.g.

```
bash$ grep ‘^Celtic ‘ results
```

will print out any lines starting with the word “Celtic” in the file results.

Useful flags to grep are “-i” for a case-insensitive search and “-v” to negate the search result i.e. in the above example it would print all the lines that don’t start with the word “Celtic”.

### awk

Awk is a full programming language in its own right and is commonly used on working with columns of data like you would get in a spreadsheet or output from a number-crunching computing job. Common tasks performed using awk are transposing columns, printing only specified columns and performing arithmetic on columns e.g. summing values in a column to calculate the total.

```
bash$ awk –F, ‘/^[Tt]oast/ {print $1 “costs” $NF }’ menu
```

### sed

Sed is the Stream Editor and operates on a stream of text. A stream can be generated from a file or by upstream filters in a pipe. Common uses are deleting lines from a stream:

```
bash$ sed '/betty boo/,$d' chart
```

or using regexes (regular expressions) to substitute patterns in a stream

```
bash$ sed 's/^fred/barney/' flintsone
```

### sort

The sort command is used to sort a file by ASCII or numerical order. As it is a filter it can also be used to sort the output from upstream filters in a pipe.

```
bash$ sort –t: -n –k2 bills
```

### Filters in Pipelines

Filters can be combined in a pipeline to perform potentially complex manipulation on data streams.

```
bash$ who | grep –v 'hacker' |awk '{print $1 is trustworthy}'
```

# What Is A Shell Script ?

- **normal text file containing shell commands**
- **should be executable and readable**
- **created using a suitable text editor e.g. nano, vi, emacs**
  - **You'll be doing this a lot in today's exercises!**

---

### Shell Script

A shell script is just a normal Unix file which contains Unix and shell commands. The simplest shell scripts simply group together commonly used sequences of commands.

More complex scripts use the shell's programming syntax to perform more advanced tasks.

Shell scripts are not compiled but interpreted. This means that each time they are run a shell is executed to read the file and run the commands it contains. Languages which are compiled, such as C, will produce a compiled code which will run faster than an equivalent shell script.

However, since shell scripts usually use other Unix commands to achieve their task, and are much easier to write, this usually does not matter. As we shall see the shell is good for some things and not so good for others.

### Execute Bit

Like any program under Unix a shell script must have the execute permission bit turned on. To do this use

```
bash$ chmod +x script-name
```

In fact, since the shell actually reads the script each time it is executed you need to turn the read permission bit on as well. In general when you create a file under Unix you are given read and write permission on the file, so you need only add execute permission.

### Text Editors

As they are essentially just text files, shell scripts are usually created using a text editor. There are a variety of text editors available on most Unix systems (emacs, vi, nano). We recommend "nano" unless you are an experienced Unix user and are comfortable with more advanced editors like emacs or vi.

## Comment and #!

- **# is comment character**
- **#!command - specifies command interpreter**

```
#!/bin/sh
echo "Hello world"
```

**Comment Character**

The comment character in the shell is the #

Any text between the # character and a newline is ignored by the shell. A comment character will only be recognised as the first character in a file or following a space, tab or newline. The comment character can of course be quoted to escape its meaning. As with all programming languages it is very important to comment your code!

**Command Interpreter**

The special directive #! as the first two characters of a file tell Unix that the rest of the line identifies the program which should be used to run this file. So to make sure a script runs under the Bourne shell use

```
#!/bin/sh
```

Note that this special directive is only recognised if it is the first two characters in the file and so there should be no spaces before the #. There can however be spaces between the ! and the name of the interpreter.

## Task 1.3 Create count-users script

Write a shell script, called count-users, which counts how many users are on the system.

( Tip – The who command shows all the users on the system - one per line. The wc -l command prints how many lines it had as input ).

## Task 1.4 Modify count-users script

Modify your script from exercise 3 above to use the date command to display the result as follows. (Tip - use command substitution to print the date and number of users on one line. )

```
At Wed Jan 18 17:58:43 GMT 1995 there are 100 users on the system
```

Change the command interpreter from /bin/sh to /bin/cat - what happens ?

# Shell variables

- **Set**
  - **Variable=value**
- **Dereference**
  - **$Variable**
- **Variable Names**
  - **Must start with an alphabetic character**
  - **May contain a-zA-Z0-9_**
  - **Are case sensitive - FOO Foo and foo are different**

---

### Shell Variables

Using shell variables should already be a familiar operation, eg setting your terminal type is done by setting the TERM shell variable. A shell variable is simply set by equating it to a value. Note that there should be no spaces either side of the = and if the variable's value contains spaces or tabs it should be quoted for example -

```
bash$ Greeting="Hello Tony"
```

A variable's value is obtained by prefixing it with the $ character, for example -

```
bash$ echo Greeting
Greeting
bash$ echo $Greeting
Hello Tony
```

### Variable Names

Variables names may comprise upper and lower case alphabetic characters, digits and underscores. A user defined variable name cannot start with a digit. As with most things in Unix variable names are case sensitive, e.g. FOO and foo are two distinct variables.

Of course the variable's value can be any string of characters!

### Special shell variables

Some variables have special significance to the shell. For example, the PATH variable is used by the shell to contain the list of directories to search for commands. Other variables have special meaning to other Unix utilities, for example the TERM variable contains the current terminal type.

# Shell variables

- **no declaration of variables**
- **variables don't have types**
- **constants via readonly**
- **unassigned variable’s value is NULL**

### No declaration, no type

Shell variables do not need to be pre-declared; in fact there is no way to declare a variable. Shell variables are created by simply assigning them. Shell variables do not have a type associated with them. Some programming languages require the type of data that the variable will hold to be declared. The shell considers all variables to contain strings of characters and so no type information is required.

### Constants

A constant, i.e. a shell variable which cannot have its value changed, can be created using the readonly command. It takes as arguments shell variable names and marks them as variables whose values cannot be altered. Trying to assign a value which has been marked as readonly causes an error and the shell script will exit.

For example -

```
Lyric="Everything is going to be alright"
readonly Lyric
```

marks Lyric as a readonly variable.

### Values of Unassigned variables

The value of an unassigned variable is an empty string which is often referred to as NULL or the NULL string.

## Default Values

- **${var}**
  - **same as $var**
- **${var-default}**
  - **If var not set produces default otherwise $var**
- **${var=default}**
  - **Same as above only var also set to "default"**
- **${var?message}**
  - **$var if var set else print "message" and exit**

---

### Default Values

The shell provides a shorthand notation for testing if a variable is already set and performing some action dependent on that test.

The simplest case is ${var-default} which tests if var is set and if not evaluates to the string default, with var being left unset. For example consider a shell script which prints a welcome message but can't be sure that the shell variable Name is set. The ${var-default} construct could be used to supply a default.

For example:

```
#!/bin/sh
echo "Hello ${Name-Sir/Madam}, have a nice day"
/usr/local/bin/dosomething
```

Note that if the default or message string includes spaces or tabs it should be quoted.

The ${var=default} could be used in the above case to also set the Name variable.

For example:

```
#!/bin/sh
echo "Hello ${Name=Sir/Madam}, have a nice day"
/usr/local/bin/dosomething
echo "Thank you $Name for doing something"
```

${var?message} is used if var being unset is a critical error and the script must exit. If

message is omitted a standard message is produced. For example, consider an application called wordprefect which needed a shell variable called WPTERM to be provided as an argument. The following script would ensure it would not run if WPTERM was not set.

```
#!/bin/sh
wordprefect ${WPTERM?"WPTERM not set, cannot run wordprefect"}
```

# Command line arguments

- **Positional Parameters** `$1 - $9`
- `shift n`
- $# **contains number of arguments**

### Command line arguments

The command line arguments which are provided to a shell script are available via the special shell variables $1 to $9. The first argument is available in the script as $1, the second as $2 until the ninth which is available as $9. These variables are often referred to as positional parameters.

For example, consider a shell script which looks to see if a friend is currently using the system.

```
#!/bin/sh
#tonyon - is tony logged in ?
who | grep tony
```

This shell script will only look for the user tony. A more useful shell script would be one which looked for any named user. A simple script to do this would be

```
#!/bin/sh
#showon - is the specifed user logged in
who | grep $1
```

### Shift

An obvious question is; how is the tenth argument obtained? The shell provides the shift command which moves, by default, each positional parameter down one place - so $2 becomes $1, $3 becomes $2 and the 10th argument becomes available as $9. Note that after this shift we have lost $1 so it should be saved to another variable before we shift !

The shift command can also take a number as an argument which specifies how many places each positional parameter should move. For example, shift 5 moves $6 to $1, $7 to $2 etc ...

### How many arguments?

$# is the shell variable which contains the number of positional parameters.

For example, a simple, and pretty pointless, script which printed out the number of command line arguments is

```
#!/bin/sh
echo I got $# command line arguments
```

If this script was called numargs, we could run it like this

```
bash$ numargs 1 2 3
I got 3 command line arguments
```

or

```
bash$ numargs hello there good to see you
I got 6 command line arguments
```

## Task 1.5 Create numargs script

Create the numargs script as above, and play with it.

Write a shell script called exon which turns the execute bit on a file given as an argument. ( Hint chmod +x filename will turn the execute bit on for filename ).

# Accessing all positional parameters

- $* **and** $@
- **"**$***"** **and** **"**$@**"**
- **"**$@**" wins !**
- $$ **- shell process id**

### Accessing all positional parameters

The shell provides two variables which contain all the positional parameters; $* and $@. When not quoted these two variables are equivalent.

The following shell script uses $* to place the contents of all the files specified on the command line in a new file called bigfile.

```
#!/bin/sh
cat $* > bigfile
```

When quoted, ie as "$*" and "$@", they are subtly different. When $* is quoted is produces a quoted string of all the positional parameters. When $@ is quoted it produces a list of quoted positional parameters. This means that when you use "$*" to refer to all the positional parameters you will lose any special quoting which has been already provided on the command line. Using "$@" will preserve this special quoting.

Consider the shell script above run with a files called "first file" and "second file". (it is possible to create files in Unix which contain spaces if you try ! ).

$* will expand to

first file second file

and so cat will have four arguments !

Using "$*" would give us

"first file second file"

and so cat would have one argument !

Using "$@" will give two arguments

"first file" and "second file"

and would produce the desired result. In short, it generally best to use "$@" to refer to all positional parameters, and the shell script above would be better written as

```
#!/bin/sh
cat "$@" > bigfile
```

### $$ process id

Another useful shell variable often used for generating a unique name for a temporary file is $$, which contains the current process id of the shell. This can be appended to a filename. The shell's process id will be between two and five digits.

For example:

```
#!/bin/sh
#ison.tmpfile
who > /tmp/wholist.$$
grep tony /tmp/wholist.$$
rm /tmp/wholist.$$
```

## Set

- **set - resets positional parameters**
- `set arg1 arg2 arg3 .....`

**Set**

The set command will reset the positional parameters to arguments it is given. If set was called as follows

```
set one two three
```

$1 would be "one", $2 would be "two" and $3 would be "three".

The other shell variables which refer to positional parameters, eg $# and $@ , and commands which act on them , eg shift, will now use the positional parameters provided as arguments to the set command.

set is often used with command substition (enclosing commands in backquotes ``) to split up the output of a command.

For example :-

```
#!/bin/sh
# timenow:
# set the positional parameters to the
# output of date
set `date`
# The fourth word is the time –
echo The time is now $4
```

On some systems the date command can actually do this selection for you.

## Task 1.6 Create last-login script

Write a shell script which uses set and command substitution to report the time you last logged in. For example:

```
bash$ ./last-login
You last logged in at 09:39
```

Tips: the "last" command lists all logins to the system, most recent first. You should filter out all users except yourself, and "head" or "sed" could be used to select the most recent login. The "whoami" command will return your own username.

# Shell Arithmetic

- **shell uses expr for integer arithmetic**
  - **num + num addition**
  - **num - num subtraction**
  - **num * num multiplication**
  - **num / num integer division**
  - **num % num remainder**
- **real arithmetic**
  - **can be kludged using bc or dc**
  - **not pretty !**

---

**expr**

The shell treats all shell variables as strings. We have to use the expr command (/bin/expr) to perform arithmetic in the shell. It takes two integer arguments and an operand and writes the result to standard output. The result from expr is usually assigned to a shell variable using command substitution.

There must be spaces between the operand and the arguments. Note that since expr uses * as the multiplication operand it must be shielded in shell scripts (usually using the \ character) as the * has special meaning to the shell.

For example, here is a shell script which will multiply two arguments:-

```
#!/bin/sh
#multiply.expr - multiply
#first arg by second
Result=`expr $1 \* $2`
echo Result of $1 \* $2 is $Result
```

expr's arguments must be integers: if given non-integer arguments it will not perform the calculation. (The isanum example script can be used to determine if a given argument is a number - the constructs used in this script will be explained later in the course).

**Real Arithmetic**

expr only provides integer arithmetic. It is possible to implement real arithmetic, ie arithmetic performed on real numbers, using either the bc or dc arithmetic utilities. Though considered slightly outwith the scope of this course a division would be performed as follows:-

```
#!/bin/sh
#divide.bc - divide first arg by
#second
Result=`echo " scale=3 ; $1 / $2 " | bc`
echo Result of $1 / $2 is $Result
```

Note: The double quotes are required to protect the semicolon which would otherwise be interpreted as a command separator.

## Task 1.7 Create Percentage Script

Write a shell script which takes two numeric arguments and calculates what percentage the first is of the second, e.g.

```
bash$ percent 1 2 50%
```

The script should use expr to calculate the percentage and so the result will be rounded to the nearest whole number.

# 2. Shell Programming

## Section 2

# Exit status

- **all commands return an exit status**
  - **zero = success**
  - **non-zero = failure**
- **man pages describe failure exit statuses**
  - **shell variable** `$?` **contains exit status of last command**
- **exit command returns status from shell-script**
  - `exit 1`

### Exit status

All Unix commands return an exit status which indicates whether the command has succeeded or failed. An exit status of 0 indicates success, whilst a non-zero status indicates failure. Some commands have more than one possible non-zero exit status depending on the nature of the failure. The grep command returns 1 if it cannot match its pattern, and 2 if is given an incorrect search pattern or cannot open the specified files. So in the example below grep can return 0 if the string "beer" is in the file shopping-list, 1 if it is not and 2 if the file shopping-list does not exist.

```
bash$ grep beer shopping-list
```

### Man page

A command's manual page should have the exit statuses that can be returned listed at the end, usually under the heading Diagnostics. For example the relevant section for grep reads :-

```
DIAGNOSTICS Exit status is 0 if any lines were selected to be printed, 1 if none, or 2 for syntax errors or inaccessible files (even if matches were found).
```

### $? = exit status

The shell variable $? contains the exit status of the last command executed. In shell scripts it is often used to test return values. The exit status is sometimes referred to as the return value.

In a pipeline $? contains the exit status of the last command, so after executing

```
bash$ who | grep tony
```

$? contains the exit status of the grep, not the who command.

### exit command

exit causes a shell script to terminate and return the specified integer value as an exit status. It is good practice always to return an exit status from your shell scripts and to follow the Unix convention of zero for success and non-zero for failure. If exit's numeric argument is omitted then the return value of the last command, i.e. $?, is used.

## Task 2.1 Examine exit status $?

Run the following commands and look at their exit statuses ( use echo $? ).

```
cat /etc/motd
cat /nofile blub
diff /etc/motd /etc/motd
diff /etc/motd /etc/hostz
diff /etc/motd /etc/hosts > /dev/null
```

# if statement

```
if condition-expression
then
    # execute these commands if condition true
    command-list
else
    # execute these commands if condition false
    command-list
fi
```

**Conditional statement**

The shell provides the "if/then/else " statement to selectively execute commands dependent on the result of a condition. If the condition is true then the commands in the "then" block are executed and if false the commands in the "else" block are executed. The whole statement is terminated by "fi".

The "else" part of this statement is optional and so statements which just test the success of the condition can be written using the following syntax :-

```
if condition
then
  command-list
fi
```

The condition which the "if" statement tests can be any set of Unix commands. When the "if" statement is evaluated those commands in the "condition" are executed and if the return value is 0 the condition is true, non-zero and the condition is false.

## if example

```
#!/bin/sh # ison - is the specified user logged in
if who | grep $1 > /dev/null
then
    echo $1 is logged in
    exit 0
else
    echo $1 is not logged in
    exit 1
fi
```

---

**The ison script**

As we have seen before "who| grep user" will test if user is logged onto the system. The ison script uses the return value of the grep in an "if" statement to present the result in a more human fashion.

If the specified user is in the list of users currently logged in then the grep will return 0 and the then part will be executed. If the grep returns non-zero, ie the user is not logged in the else part is executed.

Note that the script returns 0 if the user is logged in and 1 if not, so we can use the return value of ison in other scripts which need to look for users logged in!

**The bit bucket /dev/null**

/dev/null is a special Unix file into which output to be discarded can be directed. In the above script, the return value of the grep is of interest, not the output which it may return, and so its output is directed to the bit bucket, /dev/null.

## Task 2.2 Create doineed script using if statement

Write a shell script called doineed which searches your shopping-list file for a specified grocery item and prints

`"Yes - We are out of <item> - better get some"`

if the specified item is in the file or

`"No - we still have some <item>"`

if not in the file.

There is a sample shopping-list file provided.

## No shell negation

- **No shell negation operator to invert a condition expression**
- **Instead use : null command in if statements**

**if condition**
**then**
  **:**
**else**
  ***command-list***
**fi**

### No shell negation

The Bourne shell does not provide a negation operator to invert the sense of a conditional expression. This means there is no way to test the opposite of your conditional expression.

For example, consider the situation where an action may be required if a user is not logged in, but no action is required if they are. The conditional expression to test if a user is logged in is

```
who | grep user
```

and since the grep returns 1 ( i.e. false ) if the user is not in the who list we want to execute the else part of the if statement and have an empty then part. The shell does not allow empty command blocks.

### The : operator

The shell does provide a null command which simply provides a zero return code, this is the : command. So the above problem can be solved by writing

```
#!/bin/sh
# send-greeting - write message to
# supplied user, if logged in.
if who | grep $1 > /dev/null
then
   # User is logged in - do nothing
   :
else
   # User not logged in - exit exit 1 fi
   # We know user is logged in –
   write $1 < greeting-file
fi
exit $?
```

Here the action performed if the user is logged in is to write the contents of the greeting-file to the user's screen.

## Nesting if statements - ok !

- Nest if statements to link conditions with logical **AND**
- Alternatively use `elif` for logical **OR**

---

### Nested if Statements and elif's

It is possible to write nested "if" statements in the shell, ie the command block of a "then" or "else" part can be an "if" statement. However, when there are many nested conditional statements the code becomes hard to read and the logic unclear. The shell provides the "elif" statement as shorthand notation for "else if" statements.

Compare ...

```
if condition1
then
  command-block1 ### executes if condition1 true (condition2 not tested)
else
   if condition2
   then
      command-block2 ### executes if condition1 false AND condition2
                     ### true
   else
      command-block3 ### executes if condition1 AND conditiion2 false
   fi
fi
```

with ...

```
if condition1
then
  command-block1 ### executes if condition1 true
elif condition2
then
  command-block2 ### executes if condition1 false but condition2 true
else
   command-block3 ### executes if neither condition1 nor condition2 are
### true
fi
```

## test command

- **tests**
  - **file attributes**
  - **string comparisons**
  - **numeric comparisons**
- **returns zero if test if true, non-zero otherwise**
- **syntax**
  - **test *condition***
- **used in if statements and loops**

---

### Test command

The shell provides the "test" command to check a condition. It returns zero if the condition is true and non-zero if the condition is false. The test command provides a mechanism for checking file attributes and performing string and numeric comparisons.

It has the general syntax

```
test condition
```

For example the test condition to check if a file exists is

```
test -f file
```

test is most often used with if statements and loops, for example -

```
#!/bin/sh
# cat.silly silly shell script to test
# for a file's existence
# and cat it if it does
if test -f $1
then
   cat $1
else
   echo File $1 does not exist
fi
```

# test conditions

- **the test command can evaluate conditions for**
  - **file attributes**
  - **strings attributes**
  - **numeric comparisons**

---

### test syntax

The argument provided to test can be a shell variable or a literal string. There must be at least one space between the test command, the operator and the argument(s). The test command does not check that its arguments are valid, for example you can perform numeric tests on string value ( with unpredictable results ! ). test will produce an error if a required argument is missing, most often this happens when an unset shell variable is used in a "test" statement.

### Testing file attribute

- `test -f file ### "file" exists and is a regular file`
- `test -d file ### "file" exists and is a directory`
- `test -r file ### "file" exists and is readable`
- `test -w file ### "file" exists and is writable`
- `test -x file ### "file" exists and is executable/searchable`
- `test -s file ### "file" exists and has a size greater than zero`

"file" above actually refers to any filesystem object, eg file, directory, named pipe, block special device, etc ...

### Testing strings attributes

```
test str ### str is not null
test str1 = str2 ### str1 equals str2
test str1 != str2 ### str1 not equal to str2
```

### Testing numeric comparisons

```
test num1 -eq num2 ### true if num1 and num2 equal

also:
test n1 -ne n2 ### true if n1 not equal to n2
test n1 -gt n2 ### true if n1 greater than n2
test n1 -ge n2 ### true if n1 greater than or equal to n2
test n1 -lt n1 ### true if n1 less than n2
test n1 -le n2 ### true if n1 less than or equal to n2
```

## Task 2.3 Create filetype script using test

Write a shell script, called filetype, which takes a single argument and determines if it is a file or directory, or other in which case it exits. If a file it prints out if it is readable, writeable or executable. If a directory it prints out if it is readable, writeable or searchable.

## More test

- **! = test negation operator**
- **-a = logical and**
- **-o = logical or**
- **( ) for parenthesis**
- **[ ] = alternative test syntax**

### Negation

Unlike the shell the test command has a negation operator to invert the sense of the test. So the shell script below prints a warning message if the filename given as an argument does not exist.

```
#!/bin/sh
#cat-file.1
# Make sure we have an arg
if test $# -ne 1
then
   echo Must supply one argument
   exit 1
fi
# Make sure we can read it
if test ! -r $1
then
   echo $1 - cannot read
   exit 2
fi
# Now we can do something with file $1 –
# Let’s try cat’ing it!
cat $1
exit $?
```

### Logical "and/or" and parenthesis

The "test" command has logical "and" and "or" operators to combine conditions. The open and closing brackets, "(" and ")", can be used for grouping tests, but as these characters are significant to the shell they must be quoted.

And so the above can be written as ...

```
#!/bin/sh
#cat-file.2
if test $# -eq 1 -a \( -f "$1" -a -r "$1" \)
then
   cat $1
else
echo "Error !!"
fi
```

### Alternative test syntax [ ]

There is an alternative way of writing the "test" command using square brackets. The word "test" is replaced by a [ and the test condition is terminated by a ]. So an equivalent statement to check that a shell script has one argument is

```
#!/bin/sh
# test-arg
if [ $# != 1 ]
then
   echo Must supply one argument
   exit 1
fi
```

## Task 2.4 Create grade script

Write a shell script called grade, using numeric tests and if/elif statements, which takes a single numeric argument and prints out the corresponding grade based on these bands. Produce relevant error messages for numbers less then 0, or greater than 100.

```
A 80 – 100

B 60 -79

C 30 – 59

D 10 – 29

E 0-9
```

## case statement

- **multiway branch statement**

```
case string in
  pattern1)
      command-list
  ;;
  pattern2)
      command-list
  ;;
  patternN)
      command-list
  ;;
esac
```

---

**case statement**

The case statement matches a string against a list of patterns. If a match is found the corresponding command list is executed. Only one pattern can be selected in the case statement. If two or more patterns can be matched the first is selected and the corresponding command-list executed. If no patterns are matched no command-lists will be executed.

Each command list is terminated using two semi-colons. The pattern specification uses the shell's pattern matching facilities - * ? and [ ]. More than one pattern can be associated with a command-list using the | character to separate alternate patterns. For example the pattern

```
food|drink)
```

would match the strings "food" or "drink".

Often the last pattern in a case statement will be * which will match any string. This provides a way of having a default action in a case statement as if none of the previous patterns are matched, * as a final pattern will always be matched. In fact, since matches are performed in order and * will match any string, it should only ever be used as the last pattern !

The case statement is often clearer than a long if/elif statement, though it can only be used if the test condition can be expressed as a pattern match. So most numeric tests or tests of file attributes must be performed with if/elif statements.

# case example

```
#!/bin/sh # what-we-got - determine character type of first arg
case "$1" in
  [0-9]*)
    echo "$1 starts with a number"
    ;;
  [a-z]*|[A-Z]*)
    echo "$1 starts with an alphabetic"
    ;;
  *)
    echo "$1 starts with a non-numeric or alphabetic character"
    ;;
esac
```

**what-we-got**

This script uses a case statement to test what type of character a given argument starts with. It uses the * to provide a default action if the argument does not start with a alphabetic or numeric character.

## Task 2.5 Create inspect-file script

The "file" command takes a filename as an argument and prints out the files "type". For example,

```
bash$ file examples/ison
examples/ison: executable shell script
bash$ file /etc/passwd
/etc/passwd: ascii text
bash$ file /bin/cat
/bin/cat: ELF 32-bit MSB executable
bash$ file /var/adm/pacct
/var/adm/pacct: data
```

Write a shell script, which uses a case statement, that takes a single filename as an argument and, using "file" to determine its "type" of the above.

Call this shell script inspect-file.

( Hint: the output of the file command can be assigned to a shell variable using command substitution. This variable can be used as the "string" in a case statement, with the different actions determined by an appropriate pattern ).

"od" is a Unix utility which provides a file dump ( numeric representation of binary data ).

## Task 2.6 Bonus exercise:

Add a check for the correct number of arguments and that the argument is a regular file and readable.

If $PAGER is set use it in preference to "more", if not set default to "more".

- runs less on the file if it is "text".
- runs `od "filename"| more` on the file if it is "data" or an "executable".
- prints "unknown filetype" if neither.

# Good looking code

- **start keywords on newlines**
- **comments**
- **indentation**
- **verbose options**

### Program layout

It is easier to read and debug well-presented code. Here are some tips for writing "presentable" code:

### Keyword layout

The special words the shell uses for its conditional and loop syntax

```
if then else elif fi for until while do done case in esac
```

are called keywords. The shell's keywords are only recognised after a newline or a semicolon ( the shell's statement separator ). It is easier to read shell scripts which take a newline for each shell keyword.

### Comments

Good code is always "well" commented. In general a comment to say what the shell script does is essential.

### Indentation

Indenting code for each logical code block makes the code easier to understand.

### Verbose options

Consider using verbose options to commands which can reduce the need for verbose comments. For example, when listing files one might use `ls --reverse` rather than `ls -r`

# 3. Shell Programming

## Section 3

# Looping in the Shell

- **what is a loop ?**
- **shell loops**
  - **for**
  - **while**
  - **until**

### Loops

Loops provide a mechanism for repeating a set of commands for a list of variables or until a certain condition occurs. The shell provides three types of loops; for, while and until loops.

### for loop

The for loop performs actions on a predefined list of items.

### while loop

The while loop performs actions whilst a condition is true.

### until loop

The until loop performs actions whilst a condition is false, i.e. until it is true.

# for loop

- **General syntax**

```
for var in word-list
do
    command-list
done
```

---

### For Loop

The for loop is useful for performing the same action on a list of items. The for loop executes the command-list once for each "word" in the word- list. The loop variable, var, is assigned to the next "word" each time round the loop. Each separate "word" or item in the word-list is separated by white space. If a list item contains white space it should be quoted to prevent it from being broken down into multiple items.

The word-list can consist of

- parameters, see section 1 on "Accessing all positional parameters".
- a constant string of text ( see counting )
- filenames generated using shell pattern matching ( see diffold )
- shell variables ( see printargs )
- output from another command using command substitution ( see find- string)
- any combination of the above
- empty !

If the "in word-list" part of the for statement is omitted then the word-list is set to the current positional parameters - in most cases the script's command line arguments. It is actually set to the value of "$@", ie quoted string of all the positional parameters

# for loop examples

```
#!/bin/sh
# counting - count and wait
for i in one two three
do
      echo $i
      sleep 1
done
```

### counting

This script is an example of a for statement in its simplest form. The word-list is a constant list of strings. As the loop is processed var will be set in turn to "one" "two" and then "three". The "sleep" command causes the script to pause for the specified number of seconds.

---

```
#!/bin/sh
# remindme - static reminder list
echo "remember you have to"
for Task in "Wash the dog" "Hoover the house" "Phone Aunt Flo"
do
      echo $Task
done
```

### remindme

This is another example of a for statement with a constant word-list. This time items in the word-list have been grouped using quotes. The loop body, ie echo $Task, is executed three times with Task set to "Wash the dog", "Hoover the house" and then "Phone Aunt Flo". If all quotes were removed from this line the loop body would be executed nine times - once for every word.

```
#!/bin/sh
# filetypes - print file type of all files in current directory
# for all files ...
for file in *
do
      file $file
done
```

**filetypes**

In this example the word-list is generated using shell pattern matching. Since * will match all files, the loop body is executed once for each file in the current directory. Therefore the **file** command will be run on each file in the current directory in turn.

If the shell pattern match fails to select any files then the word-list is set to the actual shell pattern match.

For example :-

```
#!/bin/sh #echo-a-files
for File in a*
do
  echo $File
done
```

will echo each filename starting with an "a". If there are no files in the current directory it will actually echo the string "a*". It is always prudent to check the file which a shell expansion has matched using "test -f".

## Task 3.1 Create create-logs and date-to-logs scripts

Write a shell script, called create-logs, which uses a for loop to create files called log-uno log-dos log-tres in the current directory. (Tip The touch command will create a named file so

```
bash$ touch foo
```

will create a file called foo ).

Write a shell script, called date-to-logs, which appends the string

```
This command run at <date>
```

where <date> is the current date to all files in the current directory which start with the string "log-".

What happens if you make a directory called log-quatro and run the script.

Delete the "log-" files - what happens if you run this script?

Modify date-to-logs to check that the file it is trying to append to is a regular file and writeable.

## More for loop examples

```
#!/bin/sh
# print-args - print the arguments !
for Arg in "$@"
do
    echo $Arg
done
```

**print-args**

print-args uses a shell variable, "$@" which contains a quoted list of positional parameters, as the word-list in its for loop. If no arguments are given to this shell script then nothing is output, otherwise each argument is echoed back.

---

```
#!/bin/sh
# show-files - run less Numbadfiles=0
for File in "$@"
do
    if test -f $File -a -r $File
    then
        # $File is a file and readable - process it
        less $File
    else
        Badfiles="$Badfiles $File"
        Numbadfiles=` expr $Numbadfiles + 1 `
    fi
done
echo $Numbadfiles bad files - $Badfiles
```

**show-files**

Often the arguments given to a shell script are the names of files which are to be processed. show-files again uses "$@" as the word list to act on each of the positional parameters. Each argument is tested to ensure it is a file and is readable and if so the file is viewed on the screen using the pager " less". If a given argument is not a file or is not readable then its name is appended to the Badfiles variable and a count of the number of bad arguments incremented. After all arguments have been processed the number and names of bad arguments are output.

```
#!/bin/sh
# mates - are the following people logged in
Mate_list=${Mates-"rob paul gavini"}
for Mate in $Mate_list
do
    ison $Mate
done
```

**mates**

This shell script uses the ison script shown earlier to test if any of the named users are currently using the system. Here a shell variable is used as the for loop word-list. It is set to the value of the Mates environment variable (inherited from the calling shell) if set or defaults to a fixed list. ison is then run for each item in the word_list.

( Default values via ${var-default} are explained in Section 1 - Default Values. )

## Task 3.2 Create a numfiles-in-dirs script

Run the print-args script with the following arguments

```
1 2 3 4 "1 2" "2 3" `date`
```

Write a shell script, called numfiles-in-dirs which takes as arguments a list of directories and prints how many files are in each directory. This script should print "appropriate" error messages.

(Hint: "ls $Dir | wc -l" will produce a count of the number of files or directories in $Dir ).

## Final for loop example

```
#!/bin/sh
# find-string - find named string in files in cwd
# and copy them to directory of same name
mkdir $1
for File in `grep -l $1 *`
do
    cp $File $1/$File
done
```

### find-string

This is an example of using command substitution to construct the for loop's word-list. This script takes as an argument a string to search for in the files in the current directory. It creates a directory of this name and copies any files which contain this string into it. "grep -l" lists the names of files containing the string.

### Bonus Exercise

This shell script could be improved by testing the number of supplied arguments ( using an if statement to test $# = 1 ) and a test that the mkdir succeeded.

# While loop

- **General Syntax**

```
while condition
do
    command-list
done
```

---

### While loop

The while loop executes the loop body whilst the condition is true. The condition's value is its return status and so as we have seen with the if statement, a return value of zero is true and non- zero is false. Note that if the condition is never false then the loop will never exit ( unless an explicit break statement is encountered - see "Loop control" later in this section. )

Again the condition can be any series of shell statements. The condition’s value is the exit status of the last command executed in the condition. The while loop's condition is often a test command.

## while loop examples

```
#!/bin/sh
# watchforlogout-loops until specified user has
logged
# off then prints a message
while ison $1 > /dev/null
do
    sleep 60
done
echo "$1 logged off"
exit 0
```

### watchforlogout

This script uses the "ison" script shown earlier to test if the named user is logged in. If they are, ie ison returns 0, the loop body is executed and so the script sleeps for sixty seconds. Once the user has logged off ison will return false, ie non- zero, and so the loop will terminate and a message will be printed to this effect. Note again, that a redirection to /dev/null is used to discard the unwanted output from ison, as only its return value is of interest.

```
#!/bin/sh
# blast-off - countdown from ten ... then !
i=10
while test $i -ge 0
do
    echo $i
    i=`expr $i - 1`
    sleep 1
done
echo " ... we have lift off"
exit 0
```

**blast-off**

blast-off is an example of a while loop which uses expr and a numeric test to execute a list of commands a set number of times. In other programming languages this is often performed using a for loop, in the shell this construct is used.

Each time round the loop the variable tested in the loop condition is decremented. Once the variable, $i, is 0 the test is no longer true and the loop terminates.

Often loops which are executed a predetermined number of times are used to process arrays. The Bourne shell does not provide an array construct.

Contradiction: actually, though the shell does not provide arrays as a variable type ( in fact as mentioned before variables are all strings in the shell ), it is possible to "kludge" them. It is unpleasant.

```
#!/bin/sh
# wait-for-results - mail a message once
Results_file
# has been created.
Results_file="results"
Pause=60
Recipient=”Joe.Bloggs@ed.ac.uk”
while test ! -f $Results_file
do
    sleep $Pause
done
echo "Results have arrived" | /bin/mail $Recipient
exit $?
```

**wait-for-results**

wait-for-results is another example of a while loop which uses test as the loop's condition. It tests for the presence of a file and if it does not exist executes the loop body. The loop body simply simply consists of a sleep statement, in which time the desired file may arrive. Once the file has been created the test will be false and the loop will terminate.

Note that the first action in the script is to initialise the three "variables". None of the variables ever change value and their literal values could be used in place of the variables, eg

sleep 60 instead of sleep $Pause. Often shell scripts will have all the initialisation of "constant" variables at the start. This allows the scripts behaviour to be easily modified.

## Task 3.3 Create a countup script from blast-off

Use the blast-off script as a basis for a countup script which prints out the numbers from 0 to the given argument.

Modify this script to accept two numeric arguments and print out the numbers between the two, e.g.

```
bash$ countup 4 6 4 5 6
```

# Until loop

- **General Syntax**

```
until condition
do # execute commands until condition true
     command-list
done
```

---

### Until loop

The until loop is very like the while loop, the difference being that the body of the loop is executed until the condition is true rather than whilst the condition is true. So condition is evaluated and if false the loop body is executed. Once the loop condition is true the loop terminates. until loops are generally used only when the loop condition cannot be negated and a while loop used.

### watchforlogin

```
#!/bin/sh # watchforlogin - loops until specified user has logged in
# then print a message
until ison $1 > /dev/null
do
      sleep 10
done
echo "$1 has logged in"
```

This script is very similar to the watchforlogout. Using the ison script and the until loop the script loops until the the condition is true, ie the user has logged in.

## Task 3.4 Modify wait-for-results script to use until loop

Modify the "wait-for-results" script to use an until loop instead of the while loop.

## Loop control

- `break n`
  - **terminate nth enclosing loop**
- `continue n`
  - **to go next loop test ( while/until ) or loop item ( for ) in nth enclosing loop**

---

### Break

The break command causes the enclosing loop to terminate and execution to resume after the loop. If break is given an integer argument then this tells the break command how many levels of nested loop to break out of.

### wait-for-results-break

```
#!/bin/sh
# wait-for-results-break- if core file present don't wait
while test ! -f results
do
  sleep ${Pause-5} # If core file present - break out
  if test -f core
  then
    break
  fi
done
```

This script is a shortened and modified version of the wait-for-results script. Here, in the loop body, we test if a core file has been created and if so we break out of the loop. ( A core file is created when a program "crashes" and it contains a memory image of the running program ).

### Continue

The continue command causes execution to resume at the next loop iteration. When used in a for loop the next loop item will be processed, and in a while or until loop the condition will be tested.

### cat-non-cores

```
#!/bin/sh
# cat-non-cores - cat files in the cwd
# which are not called core
for file in *
do
  if test -f $file
  then
    if test $file = core
    then
      continue
    fi
    # process file !
    cat $file
  fi
done
```

This script uses a for loop to cat all files in the current directory which are not called core. The for loop uses a wildcard of * to match all entries in the current directory. The test -f ensures that only files are tested and if the file is called core the continue statement causes this file to be skipped.

# read

- `read` **"reads" a line of input from the standard input**
- **General Syntax**
  - `read` ***variable-list***
- `read` **can be used in a while loop to analyse a data stream**

**read**

The read command reads one line from standard input and assigns it to the variables named in variable-list. Most often it is used to interactively read input from the keyboard. Where possible each word in the input is assigned to a separate variable. If fewer words are input than variables then the extra variables are unset, even if they previously had a value. If more words are input than variables then each variable will contain one word except for the last variable which will contain the remainder.

Consider the script -

```
#!/bin/sh
# read-example
# initialise the variables
var1=a var1=b var3=c
# read some values
read var1 var2 var3
# print them out
echo var1 is $var1
echo var2 is $var2
echo var3 is $var3
```

Given the input:

```
one two three
```

The output is:

```
var1 is one var2 is two var3 is three
```

Given the input:

```
one two
```

The output is:

```
var1 is one var2 is two var3 is
```

and so the initial value of var3 is lost.

Finally given the input:

```
one two three four
```

The output is:

```
var1 is one var2 is two var3 is three four
```

and so var 3 is assigned to the remaining input.

Note that since all remaining input is assigned to the last variable a read with one variable will assign all input to it. For example:

```
#!/bin/sh
# read-a-line read a line
echo "enter a line"
read Line
echo Line is $Line
```

Since read takes its input from the standard input, shell redirection can cause a read to take its input from a file.

So running the above script as :-

```
bash$ read-a-line < file
```

would read the first line of file .

### return value

read returns a zero, i.e. true , return value when it is able to read some input. If no input is available the read returns a non-zero value. A non-zero value will be returned when reading from the keyboard ( i.e. undirected standard input ) if the read receives an end-of-file, most commonly bound to control-D. When reading from a file it will return non-zero when is there is no more data.

### read as a while loop test condition

The exit status of the read command can be exploited in if statements or while loops to analyse an entire stream of data. The stream can be generated from a file, or a command that produces output to stdout.

```
#!/bin/sh
#
cat $FILE | while read LINE
do
    command-list
    ### we have piped the output of “cat $file” into a while loop!
    ### the read command will exit false and cause the loop to exit
    ### when it meets EOF.
    ### The loop contents are executed as a subshell
    ### - watch your environment variables!
    ### the variable LINE contains the entire current line of $FILE
done
```

**number-file**

The example script below takes its argument, the name of a file, whose contents it writes to standard output with each line prefixed by the line number. Appropriate error messages are produced if the argument it not a file.

```
#!/bin/sh
# number-file - print out file with line numbers
if test $# -ne 1
then
  echo usage number-file file
  exit 1
fi
if test ! \( -f $1 -a -r $1 \)
then
  echo error - $1
  exit 1
fi
Linenum=1
cat $1 | while read Line
do
  echo "$Linenum $Line" Linenum=`expr $Linenum + 1`
done
exit 0
```

## Task 3.5 Create sumit script

Write a shell script, called sumit, which takes as input one number per line and prints out the total.

For example,

```
bash$ sumit 1 5 7 2 ^D Total is 15
```

( Note ^D above is assumed to be the "end of file" character - try "stty -a" ).

Modify the script to use the supplied isnum script to ignore input which is not numeric and to stop reading the input when it encounters an input line of "=".

# 4. Shell Programming
## Section 4

# eval & exec

- **parse command twice**
  `eval args`
- **execute arguments as a command and exit**
  `exec args`
- **with no args can `redirect input/output`**
  exec > /dev/null

**eval**

eval evaluates its arguments as a shell command. It is useful for causing the shell to parse and expand a command line twice; the first time when the shell reads the command line and the second time when the eval command is executed.
In a simple case,

```
#!/bin/sh # eval-input
echo -n "enter input "
read Input
eval $Input
echo Command exited $?
```

would output a prompt, read an input line and then evaluate it as a shell command. The -n argument to "echo" stops the newline being output after the text.

**exec**

exec is similar to eval in that it reads its arguments and executes them. It does not reparse the input to evaluate shell expansions as eval does. Its main difference is that the command is actually executed in place of the current shell without creating a new process - so

a script will end after the exec statement is executed.

exec with no arguments is used to redirect input or ouput. For example, the following script uses exec to redirect its standard input from the /etc/motd file.

```
#!/bin/sh
# exec-redirect
exec < /etc/motd
read Line
echo $Line
```

## Task 4.1 Run eval-input script

Try running the eval-input script with the following input:

```
echo $TERM ls -l | wc -l ls /nofile
```

The example script exec-input is similar to eval-input except that the eval has been changed to a exec. What happens when this script is run with the above input ?

# Shell Functions

- **Definition**

```
name()
{
  commands
}
```

- **called as**

```
name arg1 arg2 arg3 ...
```

- **Arguments in function are** `$1 $2 $3` **...**
- **Functions have return values, analogously to exit status**
- **Functions can be exported to the environment**

```
export –f name
```

### Shell functions

More modern versions of the shell allow the use of functions. Shell functions are not very sophisticated and are at best used for eliminating repetition within scripts rather than providing true modularisation.

Shell functions are often referred to as shell subroutines or procedures.

### Definition

A function's definition starts with the function name followed by empty parentheses. The function commands are enclosed by open and close braces ie {} . The closing brace is only recognised after a newline or semi-colon, though the clearest way of defining a function is shown above.

### Invocation

A shell function is called simply by supplying the function name, with arguments following the name. Within the function these arguments are available as positional parameters, ie $1 $2 $3 ..., and the other shell variables pertaining to positional parameters are also set, eg $# contains the number of arguments provided to the function, $@ contains all the arguments. Outside the function, in the "main" shell script body, the positional parameters values are unchanged.

All other shell variables are global, ie modification to a variable's value within a function will be visible outside that function.

### Return values

A shell function, like any Unix command, can return a numeric value to indicate its success. This is done using the return command. It takes a single numeric argument which is the function's return value. If the argument is omitted the return value of the last command executed is used. When the return statement is encountered the shell function exits and execution returns to the calling function or main body of the shell script.

A function must be defined before it is called.

## Modifying variables in functions

- `$1, $2, $3…. "$@" "$*" and $#` **are local copies private to each function**
- `set` **and** `shift` **operate on the function's private copies**
- **All other variables are global and public**

### Modifying variables in functions

Often, a shell function is required to return non- numeric values or multiple values. The return statement is intended for indicating whether the function succeeded, not for returning such values.

The shell does not allow the positional parameters which refer to the functions arguments to be altered.

The positional parameter which refers to the first argument is $1. It is nonsensical to try to assign another value to this variable ( e.g. 1="value"; ! ).

The simplest way to change a variable's value within a shell function is to refer to the variable name directly; since all variables are global, modifications to a variable's value within a function will persist outwith the function.

For example consider the code below which adds one to variable i:

```
#!/bin/sh
#addonetoi
addonetoi() {
  i=`expr $i + 1`
}
i=1
addonetoi
echo i is $i
exit 0
```

Note the variable name i is hardcoded into the shell function. To alter another variable it must be assigned to i, the function called and then its value set to i's value. Hard coding variable names into shell functions is inflexible.

More usefully a function can return its value in a known variable. For example, below the function “addonetoi” always returns the result in the variable “Added_one”. An extra assignment is required to set i to its new value, though this function can now be used with any numeric variable ( see the example file addone.useful for a more useful example).

```
#!/bin/sh
#addone
addone() {
      Added_one=`expr $1 + 1`
}
i=1
addone $i
i=$Added_one
echo i is $i
exit 0
```

## prompt_and_get

```
#!/bin/sh
prompt_and_get() {
    echo "$@" \\c
    read Resp
    if test "$Resp"
    then
        return 0
    else
        return 1
    fi
}

if prompt_and_get "Please enter your name "
then
    echo name is $Resp
else
    echo failed to get name
fi
```

**prompt_and_get**

This script uses a shell function to prompt for some information and read the response. The response is returned in the variable Resp. If no input is read then the function returns 0, otherwise it returns 1. The function takes any number of arguments which are used as the text to prompt with.

Within the function the positional parameter $@ is used to refer to the arguments and test used to ensure that a response is given.

## Task 4.2 Examine and run function example scripts

Examine and run the example shell scripts - addonetoi addone addone.useful and prompt_and_get

Write a shell script called quiz which uses a shell functions to read t or f ( standing for true or false ) in response to some given questions.

The script should use a shell-function which accepts two arguments - the first being the prompt and the second being the correct response.

If all questions are answered correctly it should print a suitable congratulation, if any questions are answered incorrectly it should print an error message and exit.

For example it could be run as follows ...

```
bash$ quiz Creamed rice is good food [ t or f ] f
You should never eat anything bigger than your head [ t or f ] t
Lager is a slimming aid [ t or f ] f
Well done - you're a foodie
```

Invent your own questions or use those in the example above.

# Signals & Traps

- **traps allow commands to be executed on receipt of a signal**
- **Syntax :**

  `trap command-list signal-list`

### Signals

Signals are one of the mechanisms used by Unix to alert a process to an event. Signals are positive integers generally in the range 1 to 32 (though different versions of Unix can extend the signal list). The most commonly used signals are

- `1 hangup`
- `2 interrupt ( ^C at keyboard )`
- `3 quit ( causes core dump )`
- `9 kill ( cannot be caught )`
- `15 terminate ( die gracefully ! )`

Signals are most commonly produced by system events ( eg logging off ! ) though they can be generated by keyboard interrupts ( eg ^C ) or using the kill command.

The kill command's general form is

```
kill -signal process-id
```

e.g.

```
bash$ kill -15 121
```

will send the terminate signal to process 121.

Of course signals can only be sent to process which belong to you.

### Traps

The trap command allows signals to be caught and specified commands executed on their receipt. The syntax of the trap command is

```
trap command-list signal-list
```

The command-list is a single argument and is best enclosed within single quotes. The signal- list is a space separated list of signals. The command-list is executed on receipt of any of the signals specified on in the signal-list.

Traps are most commonly (almost exclusively !) used in shell scripts to remove temporary or lock files when interrupted.

For example the following script removes the temporary file

```
#!/bin/sh
trap 'rm /tmp/tempfile.$$;echo terminating; exit 1' 1 2 3 15
touch /tmp/tempfile.$$
echo Created /tmp/tempfile.$$
while true
do              ### continued on next page
  echo snoozing ...
  sleep 5
done
exit 0
```

## Task 4.3 Examine trap-test script

Examine and run trap-test script and from another window send various signals ( 1 2 and 15 ) and note the difference.

## Shell Debugging Options

- **Options**
  - **-x print commands and args as executed**
  - **-v print shell input as read**
  - **-n don't run the script but check its syntax**
  - **-u treat unset variables as errors**

---

### Bugs ... yuck

Most shell script bugs are caused by typos, either of variable names or shell reserved words and these are usually easily found by scanning the code.

Since the shell is interpreted rather than compiled the way to find bugs is to run the script. Unfortunately, the shell doesn't always provide the most verbose error messages. It does provide some options which aid debugging.

### Setting options

The shell options, ( eg -x -v -n and -u ) can be invoked be in several ways -

- in a known area which can be enclosed within set commands to turn on and off the option ( set +<option> will turn the option off.

### Option explanation

-x Prints the commands and their arguments as they are executed and prefixes any commands which are executed with a "+". It doesn't show the shell statements - ie if/for/while... statements.

-v Prints the shell commands as they as read. Note that when a loop is encountered all commands until the loop terminator are read before being executed.

-n Read the shell commands but do not execute them, and report syntax errors command interpreter in the script

-u Report an error if a variable is used before being initialised. Good for spotting misspelt variable names.

Generally using -x and -v in conjunction provide the most complete information:

```
bash$ sh -xv ./buggy-script1 eg prompt$ sh -x myscript
```

Note that the debug information which the shell provides is written to standard error and can be redirected

```
bash$ sh -xv ./buggy-script1 2> tracefile
```

or

```
bash$ sh -xv ./buggy-script1 > tracefile 2>&1
```

The first option is most commonly used though the second can be useful when the aim is

to collect all output into tracefile.

## Common bugs

- **<scriptname>: No such file or directory**
- **'end of file' unexpected**
- **watch out for subshells**

---

#### <scriptname>: No such file or directory

The most likely cause of this error is the script has the command interpreter specified wrongly, ie the file which the script cannot find is the command interpreter defined after the #!

This error is often confused with

```
<scriptname>: command not found
```

which the shell reports when the given scriptname is not in the search path $PATH.

#### end-of-file unexpected

This occurs whenever the script ends before the shell encounters an expected closing statement marker, i.e. fi or done. This can occur because the statement marker has been misspelt or omitted or because an open quote ( " ' or ` ) has not been closed.

#### Subshells

Under some circumstances the shell will invoke a subshell to process part of the shell script. Any variables whose values are altered in this part of the script will have these changes lost. This usually happens when a loop has its input or output redirected. To work around this problem the variables which are altered in the loop can have their values written to temporary files which are later read ( messy ! ).

## Task 4.4 Locating bugs in scripts

Inspect buggy-script1 buggy-script2 and buggy-script3, locate the errors using the techniques described and correct them.

## Conclusions

- **Shell good for**
  - **acting on filesystem**
  - **harnessing unix utilities**
  - **grouping often used sequences of commands**
- **Not so good for**
  - **numerical analysis**
  - **anything requiring arrays**
  - **"large" programming tasks**

### Shell Good ...

As we have seen the shell is a powerful programming language as well as an interactive command interpreter. The programming constructs are designed to interface well with the filesystem, e.g. wild-cards, and default arguments in the for loop. The real power of the shell is in linking small specific tools ( sort, grep etc ... ) to create larger tools

### .... But Not Always ...

There are some tasks, however, which are beyond the shell. The shell does not provide any tools for handling numerical analysis and those utilities can be called upon to do so are often unwieldly ( eg bc ! ). The shell does not provide any complex data structures ( in fact it views everything as a string ) so any programming task requiring arrays or structures/records are impossible.

The shell is not "good" for large programming tasks. The shell functions do not provide true modularisation and again the data types are missing. If truth be known most installations will have large shell scripts which have either evolved to their current size or have been written primarily to schedule commands and which have to perform many filesystem operations ...

```
bash# head -1 /usr/local/etc/backups/ backup.sh
#!/usr/local/utils/bin/bash
```

```
bash# wc -l /usr/local/etc/backups/ backup.sh 911
/usr/local/etc/backups/backup.sh
```

### ... and so Onwards ...

Where the shell fails one of the many other programming languages available for Unix will undoubtedly succeed; IS currently runs a 1 day Perl course and a 2 day Fortran course.